SPECIFICATION

Electronic Version 1.2.8

Stylesheet Version 1.0

[SMART INTERNETWORKING OPERATING SYSTEM FOR LOW COMPUTATIONAL POWER MICROPROCESSORS]

Background of Invention

[0001] BACKGROUND-FIELD OF INVENTION

[0002] The present invention relates to the field of operating systems, specifically to multitasking operating systems embedded into low processing power microprocessors.

[0003] BACKGROUND-PRIOR ART

[0004] The need for tools that facilitate programming of microprocessors has motivated the development of several operating systems that, in general, accomplish the goal of decoupling program development from details specific to the management of microprocessor hardware. Existing operating systems have rapidly evolved from simple systems with little stability and resource management capability into complex and efficient systems capable of managing several simultaneous tasks and administrating a vast number of resources. Today, it is almost impossible to imagine the existence of a personal computer (PC) that executes applications without the help of an underlying operating system.

[0005]

In addition, parallel to the evolution of PC operating systems, low-level architectures have also evolved in a way to allow the existence of low cost processors with processing speeds over 30MHz, handling several Kbytes of RAM and ROM memory, etc., all in a very small packaging. These small processors, also called

microprocessors, sometimes even include embedded peripheral devices within the casing of the microprocessor, making them ideal to solve automation, control, basic signal processing and other applications at a very low cost.

[0006] Yet, despite the great advances in microprocessor technology, there has not been a similar evolution in the development of software for these devices. It is true that there exist numerous programming tools, such as assembler language compilers, high-level language compilers with assembler output, microprocessor native language development environments and others. However, there is a growing need for a tool that facilitates rapid and efficient application development.

[0007] The use of an embedded operating system may accelerate application development by dividing the microprocessor"s operation management into specific function calls that lead to the accomplishment of these delicate tasks.

[0008] A common source of programming errors resides in the handling of microprocessor bits and registers. In operating system-based programming, the need for low-level handling is eliminated, since the operating system is now responsible of those tasks. Application development time is thus reduced.

[0009] Further, in some situations it is desirable that two or more tasks be executed concurrently, whether to be able to asynchronously handle several external events or other reasons. This requires the use of a computational resource that may allow a CPU to be shared among many tasks. Through programming based on a multitasking operating system (OS), executing multiple concurrent tasks would be as easy as developing each specific task and telling the OS to handle their execution. The idea of a multitasking operating system is not at all easy to realize, and no existing multitasking operating system supports the use of low processing power microprocessors comprising a basic architecture.

Summary of Invention

[0010] This invention presents the definition of the architecture for a multitasking operating system capable of executing on the majority of 8-bit microprocessors as well as in any microprocessor of higher processing power. Such operating system, called Smart Internetworking Operating System (SIOS), has been designed to manage

microprocessor resources, such as RAM/ROM memory, input/output ports, peripherals, and others. In addition, it is capable of handling the multiple tasks within a single central processing unit (CPU).

[0011] The definition exposed in this disclosure is based on the basic structures of the SIOS architecture, such as memory distribution, basic SIOS execution algorithms and the basic function prototypes that SIOS supports.

[0012] OBJECTS AND ADVANTAGES OF INVENTION

[0013] The following are several objects of the present invention

[0014] : · To describe a valid architecture that facilitates the implementation of a Smart Internetworking Operating System (SIOS);

[0015] · To provide a definition of the basic algorithms of the operation of SIOS;

[0016] · To illustrate and provide conceptual bases, through examples, to allow the creation of new functions that SIOS may execute;

[0017] · To provide an operating system that may be executed by low processing power and capacity microprocessors;

[0018] · To provide a software platform that permits the easy portability of application program code among several microprocessor architectures, in which each microprocessor possesses a version of SIOS adapted to its architecture;

[0019] · To provide a software platform that allows rapid development of microprocessor applications;

[0020] · To provide a platform that allows the implementation of applications comprising multiple concurrent threads of execution in a single microprocessor.

[0021] The following are several advantages of the present invention:

[0022] · SIOS is an operating system that supports low processing power microprocessors;

[0023] · SIOS provides a software development platform for microprocessors in which

program code may be ported among different microprocessors architectures;

[0024] · SIOS makes it possible to implement multi-threaded applications in low processing power microprocessors;

[0025] · SIOS supports the implementation of almost any application in a microprocessor with minimized development time, in which the limits of applications may be imposed by the intrinsic limitations of the underlying microprocessor resources (e.g., speed), and not by development time and difficulty of programming;

[0026] · SIOS allows the development of applications in which associated tasks may operate independently of each other, while sharing the same execution time and hardware resources, such as memory, etc.

Brief Description of Drawings

[0027] Fig. 1 *Data Memory - Task Control Block TCB:* A block used to control the execution of each task, called Task Control Block (TCB). Every task comprises one such block, and all TCBs are arranged in a dynamic chain of TCBs.

[0028] Fig. 2 *Data Memory - Task Information Block TIB* : Task Information Blocks (TIBs) used to control events associated with a task. Every task comprises one such block, and all RIBs are arranged in a dynamic chain of TIBs.

[0029] Fig. 3 *Data Memory - Event Control Block ECB* : ECBs control the events a system must monitor. Every event comprises one such block. All ECBs are arranged in a dynamic ECB chain.

[0030] Fig. 4 *Data Memory - Pipe Control Block PCB* : PCBs control a task's communication Pipes. Each Pipe comprises one such block. All PCBs are arranged in a dynamic chain.

[0031] Fig. 5 *Data Memory - Data Memory Control Block DMCB* : DMCBs help manage the memory assigned to a task, and are arranged in a chain of DMCBs.

[0032] Fig. 6 *Data Memory - Task Data Memory TDM* : TDM blocks are data memory blocks associated with each task. A task's TDM stores its context on one TDM block before yielding CPU handling to the Kernel.

[0033] Fig. 7 *Data Memory - Port Information Block* : PIBs are special memory blocks used to handle a microprocessor"s input and output ports.

[0034] Fig. 8 *Data Memory - Kernel Control Registers* : KCRs are registers used by the Kernel to store required operation data.

[0035] Fig. 9 *Program Memory - Task Allocation Table* : TATs are tables of pointers that accurately identify the beginning of the memory blocks associated with a task.

[0036] Fig. 10 *Program Memory - Task Header* : A task's program code section starts with a Task Header (TH), which provides basic information about the task.

[0037] Fig. 11 *Kernel Dispatcher General Flow Chart* : The general algorithm of the Dispatcher is described later, with the figures that illustrate every stage in the general algorithm.

[0038] Fig. 12 *Kernel Port State Update* : This control block is the first step of four that the Dispatcher executes before yielding control to the active task. This subprocess carries out reading and verification of a microprocessor"s input ports to set up port events, which can be signaled when the state of an input port changes.

[0039] Fig. 13 *Kernel Task State Update* : This control block is the second step that the Dispatcher executes before yielding control to the active task. This subprocess carries out reading and verification of events to update the state of tasks on the WAIT state and switch them to the READY state if their expected event is signaled.

[0040] Fig. 14 *Kernel Priority Task Ordering* : This control block is the third step of four that the Dispatcher executes before yielding control to the active task. This subprocess selects the task in the READY state that is to be switched to the ACTIVE state.

[0041] Fig. 15 *Kernel Context and Control Restore* : This control block is the last step that the Dispatcher executes before yielding control to the active task. This subprocess executes the restoration of all context variables associated with the active task, and yields CPU control to the Parser, which continues the execution of the active task's instructions.

Detailed Description

[0042] DETAILED DESCRIPTION OF DRAWINGSFig. 1 *Data Memory - Task Control Block TCB:* This Data Memory block, called TCB, is used in the execution of each task. Every task comprises one such block. All TCBs are arranged in a chain of TCBs, and contain the following fields: · Task_ID: Task Identifier, contains the identifier of the task who owns a TCB. This field is initialized when a task is installed into the Kernel"s line of execution, and is never modified.

[0043] · Status/Priority: Contains the task's state and priority information. Bit<7..6> correspond to the state. Bit <5..0> correspond to the priority. A task's priority is fixed and is obtained from the Task Header when the task is installed on the Kernel"s line of execution. The state may be one of the following: 00 for ACTIVE, 01 for WAITING, 10 for READY, or 11 for SUSPEND.

[0044] · Task_Next_IP: Task Next Instruction Pointer is a set of three bytes that store the pointer to Program Memory where the next SIOS instruction associated with the task that owns the TCB resides. Bytes are assigned in the order of High, Medium and Low, where Low byte is the least significant byte, and High byte is the most significant. This pointer is updated every time a task yields CPU control to the Kernel.

[0045] · Task_Start_IP: Task Start Instruction Pointer is a set of three bytes that contain the pointer to Program Memory where the first SIOS instruction associated with the task that owns the TCB resides. Bytes are assigned in the order of High, Medium and Low, where Low byte is the least significant byte, and High byte is the most significant. This pointer is installed in the Kernel"s line of execution with the value indicated by the Task Header, and is never modified.

[0046] · Ready_Wait_Time: Task waiting time in Ready State, byte that contains the counter of how many times a READY task has lost the priority competition. This value is updated by the Kernel.

[0047] · Event_ID: Event Identifier contains the event identifier for which a task is waiting to switch from the WAIT to the READY state. This value is updated when the task wishes to wait for an event.

[0048] · Event_Control: Event Control byte contains the event control value that an event

must display so that it is assumed as active and a task may continue operation. This value is updated when a task starts its wait for an event.

[0049] · Next_TCB: Next Task Control Block Pointer is a set of two bytes that contain the pointer to Data Memory of the next TCB in the chain of TCBs. If NULL, it means that there are not other TCBs on the chain. Byte order is big-endian (i.e., the first byte is the most significant byte, and the last is the least significant byte). This value is updated by the Kernel when the current TCB is the last on the chain and a new TCB is created.

[0050] Fig. 2 *Data Memory - Task Information Block TIB*: A TIB is used to control the events associated with a task. Every task comprises one such block. A TIB comprises the following fields: · Task_ID: Task Identifier contains the identifier corresponding to the task that owns a TIB. This field is initialized when the TIB is created, when the task is installed in the Kernel"s line of execution, and is never modified.

[0051] · Pipe_ID: Pipe Identifier contains a pointer to the Pipe that a task uses for communication purposes. If NULL, it means the task has not been assigned a Pipe. This field is initialized when a task requests that a Pipe be assigned to it.

[0052] · Mutex_ID: Mutual Exclusion Identifier contains the identifier of the mutually exclusive event (Mutex) that a task owns. If NULL, no mutex has been assigned to the task. This field is initialized when a task requests that a mutex be assigned to it.

[0053] · Next_TIB: Next Task Information Block Pointer is a set of two bytes that contain a data memory pointer to the next TIB in the TIB chain. If NULL, it means no next TIB exists in the TIB chain. Byte order is big-endian. This value is updated by the Kernel when the current TIB is the last on the TIB chain and a new TIB is created.

[0054] Fig. 3 *Data Memory - Event Control Block ECB*: This block is used for control of events that the system is to monitor. Each event comprises one such block. An ECB consists of the following fields: · Event_ID: Event Identifier contains the identifier of the event being expected by a task so that it can switch from the WAIT to the READY task state. This field is initialized when an event is created and is never modified.

[0055] · Control/Status: An event"s Control & status contains information about the

event, whether the event is signaled or not, and other event data. This field is updated by the task that controls the event. The Kernel watches this field to decide whether an event has been signaled by comparing it with the event"s control byte Event_Control within the TCB.

[0056] · Next_ECB: Next Event Control Block Pointer is a set of two bytes that contain the data memory pointer to the next ECB in the ECB chain. If NULL, it means there is no next ECB in the chain. Byte order is big-endian. This value is updated by the Kernel when the current ECB is the last on the ECB chain and a new ECB is created.

[0057] Fig. 4 *Data Memory - Pipe Control Block PCB*: The PCB is used for Pipe control, and there is one PCB for each existing Pipe on the PCB chain. The PCB contains the following fields: · Pipe_ID: Pipe Identifier contains the identifier of the Pipe created on memory. This field is initialized when the Pipe is created and is never modified.

[0058] · Pipe_Base_Address: Pipe Data Memory Base Address is a data memory pointer to the first byte on the Pipe. This field is initialized when the Pipe is created and is never modified. Byte order is big-endian.

[0059] · Pipe_Size: Pipe Size in bytes, contains the number of data memory bytes that a Pipe occupies. This field is initialized when the Pipe is created and is never modified.

[0060] · Next_PCB: Next Pipe Control Block Pointer is a set of two bytes that contain a data memory pointer to the next PCB in the chain of PCBs. If NULL, it means there is no next PCB on the chain. Byte order is big-endian. This value is updated by the Kernel when the current PCB is the last on the PCB chain and a new PCB is created.

[0061] Fig. 5 *Data Memory - Data Memory Control Block DMCB*: A DMCB is used to manage the memory assigned to each task. DMCBs are arranged in a chain of DMCBs, in which each DMCB contains the following fields: · Task_ID: Task Identifier contains the identifier of the task that owns the DMCB. This field is initialized when the block is created, when the task is installed on the Kernel"s line of execution, and is never modified.

[0062] · Data_Memory_Base_Address: Task Data Memory Base Address is a data memory pointer to the first byte in the memory block assigned to a task. These bytes are

initialized when Task Memory Data is assigned and are never modified. Byte order is big-endian.

[0063] · Data_Memory_Size: Task Data Memory Size in bytes, contains the number of data memory bytes that the assigned memory block takes up. This field is initialized when the block is created and is never modified.

[0064] · Next_DMCB: Next Task Data Memory Control Block Pointer is a set of two bytes that contain a data memory pointer to the next DMCB in the DMCB chain. If NULL, it means no next DMCB exists. Byte order is big-endian. This value is updated by the kernel when the current DMCB is the last DMCB on the DMCB chain and a new DMCB is created.

[0065] Fig. 6 *Data Memory - Task Data Memory TDM* : Every task comprises a data memory block containing Task Data Memory (TDM) to enable it to store the context registers before yielding control to the Kernel, and the registers required for regular operation. TDMs contain the following fields: · Task_ID: Task Identifier contains the identifier of the task that owns this TDM. This field is initialized when the block is created, when the task is installed on the Kernel"s line of execution, and is never modified.

[0066] · REGx: task context register. This may be a STATUS register of the ALU, an INDIRECTION register, a bank selection register, or other, and contains the value of the REGx before yielding control to the Kernel. This value is accordingly recovered from this location after control is resumed. There are as many REGs as required by the system for successful context change.

[0067] · DATAn: task data registers. There are as many assigned registers as the task requested when it was installed on the Kernel"s line of execution.

[0068] Fig. 7 *Data Memory - Port Information Block* : PIBs are special data memory blocks used for handling of microprocessor input and output ports. They contain the following fields: · Assignation_Port_Mask: Mask comprising a set of bytes that contain bitwise information of microprocessor inputs/outputs (I/Os) assigned to tasks. If the I/O bit is HIGH, it means that the I/O has been assigned to a task. If the bit is LOW, it is free to be assigned to a task. There are as many bytes in the mask as there are data

bytes provided for I/Os by the microprocessor. Byte order is little-endian (i.e., the first byte is the least significant whereas the last byte is the most significant).

[0069] · Input_Selection_Mask: Mask comprising a set of bytes that contain bitwise information of the direction of communications in I/Os. If the bit is HIGH, the I/O is being used as input and must be watched to verify that it generates a port event. Otherwise, it is being used as output and does not have to be watched. There are as many bytes in the mask as there are data bytes provided for I/Os by the microprocessor. Byte order is little-endian.

[0070] · Idle_State_Mask: Mask comprising a set of bytes that contain bitwise information about the natural state of microprocessor I/O bits. If a bit is HIGH, it means that a current LOW value must generate the port event. If a bit is LOW, it means that a current HIGH value must generate the port event. There are as many bytes in the mask as there are data bytes provided for I/Os by the microprocessor. Byte order is little-endian.

[0071] · Change_State: contains a set of bytes that contain bitwise information related to a change in the value of the microprocessor"s I/O bits. If the bit is HIGH, it means that a change has occurred in such bit. If a bit is LOW, a change in such bit has not occurred. There are as many bytes in the mask as there are data bytes provided for I/Os by the microprocessor. Byte order is little-endian.

[0072] Fig. 8 *Data Memory - Kernel Control Registers* : The Kernel requires a set of information storage registers for successful operation. This block of data memory contains the following fields: · DM_Remainder: Free Data Memory Remainder, contains the number of free bytes in data memory. Every time a task is assigned a memory block, this counter is decreased. Byte order is big-endian.

[0073] · Max_Priority: register that stores the maximum priority assigned to a task used in the process of searching for the READY task with highest priority.

[0074] · Max_Wait_Time: register that stores the maximum wait time associated with a task. This is used in the process of searching for the READY task with the highest priority.

[0075] · Task_ID_Winner: Identifier of the READY task of highest priority.

[0076] · IP: Instruction Pointer, pointer to program memory that contains the current instruction that is to be decoded by the Parser. Byte order is big-endian.

[0077] · DM_Pointer: Data Memory Pointer. Byte order is big-endian.

[0078] · TCB_Pointer: Task Control Block Pointer contains a data memory pointer to the location of the current TCB. Byte order is big-endian.

[0079] · ECB_Pointer: Event Control Block Pointer contains a data memory pointer to the location of the current ECB. Byte order is big-endian.

[0080] · DMCB_Pointer: Data Memory Control Block Pointer contains a data memory pointer to the location of the current DMCB. Byte order is big-endian.

[0081] · TIB_Pointer: Task Information Block Pointer contains a data memory pointer to the current TIB. Byte order is big-endian.

[0082] · PCB_Pointer: Pipe Control Block Pointer contains a data memory pointer to the location of the current PCB. Byte order is big-endian.

[0083] · Last_TCB_Pointer: Last Task Control Block Pointer contains a data memory pointer to the location of the last TCB in the TCB chain. Byte order is big-endian.

[0084] · Last_ECB_Pointer: Last Event Control Block Pointer contains a data memory pointer to the location of the last ECB in the ECB chain. Byte order is big-endian.

[0085] · Last_TIB_Pointer: Last Task Information Block Pointer contains a data memory pointer to the location of the last TIB on the TIB chain. Byte order is big-endian.

[0086] · Last_PCB_Pointer: Last Pipe Control Block Pointer contains a data memory pointer to the location of the last PCB on the PCB chain. Byte order is big-endian.

[0087] · Last_DMCB_Pointer: Last Data Memory Control Block Pointer contains a data memory pointer to the location of the last DMCB in the DMCB chain. Byte order is big-endian.

[0088] Fig. 9 *Program Memory - Task Allocation Table* : A Task Allocation Table is a set

of pointers used to keep accurate track of the beginning of each memory block associated with a task's program code. TATs are read by the system initialization process. There are as many pointers in TATs as there are tasks installed in Program Memory. Each entry on a TAT consists of the following fields: · Task_Condition: This is a byte that indicates the initial operation condition associated with the task to which this TAT entry points. The condition may be SLEEP, to indicate that the task is not to be inserted in the Kernel"s line of execution at system start time. Or it may be WAKE, to indicate that the task must be installed on the Kernel"s line of execution upon system startup.

[0089] · Task_Start_IP: Task Start Instruction Pointer is a set of three bytes that contain a program memory pointer to the location of a task's Task Header. Byte order is high, medium and low, where HIGH is the most significant byte, and LOW is the least significant byte.

[0090] Fig. 10 *Program Memory - Task Header* : A task's program code section starts with a Task Header (TH) that provides the system with basic information about the task. The following are the field contained in a Task Header: · Task_ID: Task Identifier contains the identifier of the task that owns this TH.

[0091] · Status/Priority: initially contains the task's state information and assigned priority. Bits<7..6> correspond to state, and Bits<5..0> correspond to priority. State can be 11 for SUSPEND, or 10 for READY.

[0092] · Task_Data_Memory_Size: Task Data Memory Size in bytes, contains the number of bytes that the assigned data memory block occupies.

[0093] · Task Program Code: is the task's program code that is to be executed by the Kernel.

[0094] Fig. 11 *Kernel Dispatcher General Flow Chart* : This is the general algorithm performed by the Dispatcher and is described in detailed on the next figures.

[0095] Fig. 12 *Kernel Port State Update* : This control block is the first step of four steps carried out by the Dispatcher before yielding control to the active task. This subprocess includes reading and verification of input ports to the microprocessor, so

that the appropriate event can be signaled when a change is detected on the associated input ports. Verification is done during every cycle in which the Dispatcher takes control. This high frequency sensing enables the OS to detect events in high speed ports, achieving the best possible resolution.

[0096] The result is a mask of assignment and selection of input ports, which details the ports that must be monitored and those that need not be monitored. Ports that must be monitored are those that have been assigned to tasks as input ports. Ports that need not be monitored are those not assigned to any tasks, or assigned to tasks as output ports. If the input port events are active, they are reset.

[0097] Next, the resulting mask is examined. If it is all-zeros, then no port verification is necessary as not bits are programmed to generate events. All microprocessor ports are checked, and the result is masked with the mask resulting of block A to obtain the real value of the bits that are to be watched.

[0098] Finally, bit changes are detected by checking the current state mask with the mask corresponding to the natural state of bits. If a change occurred, the associated bit will be HIGH, otherwise it will be LOW. If the resulting mask is all-zeros, no changes occurred. Otherwise, it means that there were changes in the input ports. Accordingly, the event of Port_Change_Event is signaled.

[0099] Fig. 13 *Kernel Task State Update* : This control block is the second step that the Dispatcher must carry out before yielding control to the active task. This subprocess includes reading and verification of events, so that tasks in the WAIT state can be switched to the READY state if their expected events have occurred.

[0100] First, the pointer to the beginning of the TCB chain is retrieved. All TCBs in the TCB chain must be examined to acknowledge the state associated with every task. If a task is in the SUSPEND state, its state is not modified in this subprocess. The pointer to the next link in the TCB chain is obtained. If NULL, it means the entire TCB chain has been traversed.

[0101] Otherwise, the current task's state is checked. If READY, this means that the current task is not expecting any events. Its state is not modified. If WAIT, its associated event must be examined. Accordingly, the pointer to the beginning of the

ECB chain is retrieved. All ECBs must be examined until the expected event is found. If the current ECB corresponds to the expected event, its state is examined. If it is signaled (activated), the state of the task in WAIT state must be updated to the READY state. If it is not signaled, the task's state is not modified.

[0102] If the end of the ECB chain is reached without finding the associated event, the system error flag is raised, indicating that a task is waiting for a non-existent event.

[0103] Fig. 14 *Kernel Priority Task Ordering* : This control block is the third step of four steps that the Dispatched carries out before yielding control to the active task. This subprocess selects the READY task that is to be switched to the ACTIVE state and given control of the system. The criteria used for task selection are task priority and the amount of time in which a task has been in the READY state.

[0104] First, the pointer to the beginning of the TCB chain is retrieved. The entire TCB chain must be traversed to examine all tasks that are in the READY state. The Max_Priority, Max_Wait_Time and Task_ID_Winner variables are initialized. These variables will contain the information of the task that wins the competition.

[0105] All task states are checked sequentially. If a task is not READY, the next task in the chain is checked. Once a READY task is found, its priority is checked against Max_Priority. If the found task's priority is higher, its associated information is stored in Max_Priority, Max_Wait_Time and Task_ID_Winner. Such task is considered the winner, thus far.

[0106] If the priority of the examined task is equal to Max_Priority, its time on the WAIT state is compared to Max_Wait_Time. If the current task's time on the WAIT state is higher, the values of Max_Wait_Time and Task_ID_Winner are updated to reflect the newly found winning task.

[0107] These steps are repeated until the last link in the TCB chain is found (i.e., when the TCB pointer equals NULL). The winning task is the one that is registered in the Task_ID_Winner register.

[0108] Fig. 15 *Kernel Context and Control Restore* : This control block is the last step of four that the Dispatcher must execute before yielding control to the active task. This

subprocess restores the winning task's context variables and grants CPU control to the Parser. The Parser will resume execution of the active task's instructions.

[0109] First, the value of Task_ID_Winner is checked. If Task_ID_Winner is NULL, there are no active tasks to be executed. The Dispatcher thus maintains control.

[0110] If Task_ID_Winner is valid, a pointer to the start of the TCB chain is retrieved. The winner task's TCB must be found by traversing the TCB chain. If the end of the TCB chain is found (i.e., pointer equals NULL), the system error flag is raised indicating that the winner task's TCB does not exist.

[0111] If a valid TCB is found, the Parser"s Instruction Pointer is assigned with the location of the winner task's next instruction to be executed. The task's state is switched to ACTIVE.

[0112] The data memory pointer is assigned to point to the beginning of the DMCB chain to search for the active task's DMCB by traversing the DMCB chain. Once the active task's DMCB is found, the data memory pointer is loaded with the active task's DMCB. (If NULL, it means that the end of the DMCB chain was found and no valid DMCB was found for the active task. Accordingly, the system error flag is raised to indicate that the active task does not own a valid DMCB.)Next, all context variables associated with the active task are restored. The Parser now gains control and starts executing the instructions beginning with the location pointed at by IP.

[0113] OPERATION OF INVENTION

[0114] SIOS Basic Structure Data Memory Structure Data Memory is organized in such a manner as to allow the Kernel and all tasks to coexist and operate correctly. It comprises two parts: a series of control block chains and a series of data memory blocks.

[0115] SIOS Basic Structure Program Memory Structure Program Memory is the memory that contains SIOS"s program code. It comprises two sections: Task Allocation Table and Task Program Code.

[0116] SIOS Basic Structure Basic Functions: SIOS"s basic functions are those that create actions required for the existence of the operating system, for its operation and the

operation of tasks.

[0117] Basic functions are implemented in a language native to the underlying microprocessor. The function prototypes, the parameter it takes, and the values returned, are defined independently of the platform, thus allowing that code written for SIOS be easily ported to other platforms that use SIOS as their operating system.

[0118] SIOS basic functions are described next:

[0119] *SIOS_Add_Task(TAT)* This function adds a new task to the Kernel"s execution process and assigns all data memory resources needed to execute the new task.

[0120] Parameters:-TAT.Task_Start_IP, pointer to actual task to be added, in the System_Pipe[0..PM_POINTER_SIZE-1]Returns: (nothing)Actions:-Creates a TCBoVerifies if there are TCB_SIZE bytes available in memory.

[0121] oTCB_Pointer[0.. DM_POINTER_SIZE-1] = requests TCB_SIZE bytes of memory.

[0122] -Initializes TCB with necessary values: oTCB.Task_ID = Task_IDoTCB.Stat_Prio = TH.Stat_PriooLast_TCB.Next_TCB[0.. DM_POINTER_SIZE-1] = TCB_Pointer[0.. DM_POINTER_SIZE-1]oTCB.Next_TCB[0.. DM_POINTER_SIZE-1] = NULLoLast_TCB[0.. DM_POINTER_SIZE-1] = TCB_Pointer[0.. DM_POINTER_SIZE-1]oTCB.Task_Start_IP [0..PM_POINTER_SIZE-1] = TAT[Task_ID].Task_Start_IP[0..PM_POINTER_SIZE-1] oTCB.Task_Next_IP[0..PM_POINTER_SIZE-1] = TCB.Task_Start_IP[0..PM_POINTER_SIZE-1]oTCB.Ready_Wait_Time = 0oTCB.Event_ID=NULL-Creates a DMCB and a TDMoVerifies if there are DMCB_SIZE bytes available in memory.

[0123] oVerifies if there are TH.TDM_Size bytes available in memory.

[0124] oDMCB_Pointer[0.. DM_POINTER_SIZE-1] = Requests DMCB_SIZE bytes of memory.

[0125] oDMCB.Data_Memory_Base_Address[0.. DM_POINTER_SIZE-1] = Requests TH.TDM_Size bytes of memory.

[0126] -Initializes DMCB and TDMoDMCB.Task_ID = Task_IDoDMCB.Data_Memory_Size = TH.TDM_SizeoLast_DMCB.Next_DMCB[0.. DM_POINTER_SIZE-1] = DMCB_Pointer[0.. DM_POINTER_SIZE-1]oLast_DMCB[0.. DM_POINTER_SIZE-1] = DMCB_Pointer[0.. DM_POINTER_SIZE-1]oTDM.Task_ID = Task_ID-Creates a TIBoVerifies that there are

TIB_SIZE bytes available in memory.

[0127] oTIB_Pointer[0.. DM_POINTER_SIZE-1] = Requests TIB_SIZE bytes of memory.

[0128] -Initializes TIB with required values:oTIB.Task_ID = Task_IDoTIB.Mutex_ID = NULLoTIB.Pipe_ID = NULLoLast_TIB.Next_TIB[0.. DM_POINTER_SIZE-1] = TIB_Pointer [0.. DM_POINTER_SIZE-1]oLast_TIB[0.. DM_POINTER_SIZE-1] = TIB_Pointer[0.. DM_POINTER_SIZE-1]

[0129] *SIOS_Suspend_Task()* This function suspends the execution of a task. All task control blocks remain valid, i.e., TCB, ECB, TIB, PIP, TDM.

[0130] Parameters: (none)Returns: (nothing)Actions:-Searches for TCBoTCB_Pointer = FIRST_TCB_POINTERoVerify that TCB.Task_ID = Task_IDolf not, TCB_Pointer = TCB.Next_TCBoRepeat search until appropriate TCB is found.

[0131] -Updates TCBoTCB.Status = SUSPEND-Yields controloTDM.Context_Regs = Context_RegsoGoto KERNEL

[0132] *SIOS_Wake_Task(Task_ID)* This function restores execution of a suspended task.

[0133] Parameters:-Task_ID, in System_Pipe[0]Returns: (nothing)Actions:-Searches for TCBoTCB_Pointer = FIRST_TCB_POINTERoVerify that TCB.Task_ID = Task_IDolf not, TCB_Pointer = TCB.Next_TCBoRepeat search until appropriate TCB is found.

[0134] -Updates TCBoTCB.Status = READY-Yields controloGoto KERNELSIOS_Redispatch() This function yields CPU control to the Kernel. It is called by the task that wishes to yield control. The task's state is switched to READY.

[0135] Parameters: (none)Returns: (nothing)Actions:-Searches for TCBoTCB_Pointer = FIRST_TCB_POINTERoCheck that TCB.Task_ID = Task_IDolf not, TCB_Pointer = TCB.Next_TCBoRepeat search until appropriate TCB is found..

[0136] -Updates TCBoTCB.Status = READY-Yields controloTDM.Context_Regs = Context_RegsoGoto KERNEL

[0137] *SIOS_Wait_Event(Event_ID, Event_Control)* This functions switches a task into the WAIT state to wait for the specified event.

[0138] Parameters:-Event_ID, in System_Pipe[0]-Event_Control, in System_Pipe[1]. The event is assumed active when its value equals Event_Control.

[0139] Returns: (nothing)Actions:-Searches for TCBoTCB_Pointer = FIRST_TCB_POINTERoCheck that TCB.Task_ID = Task_IDolf not, TCB_Pointer = TCB.Next_TCBoRepeat search until appropriate TCB is found..

[0140] -Updates TCBoTCB.Status = WAIToTCB.Event_ID = Event_IDoTCB.Event_Control = Event_Control - Yields controloTDM.Context_Regs = Context_RegsoGoto KERNEL

[0141] *SIOS_Create_Mutex(Task_ID)* This function creates a Mutex in memory and updates the required variables.

[0142] Parameters:-Task_ID, the task that owns the mutex, in System_Pipe[0]Returns:-Mutex_ID, in System_Pipe[1] Actions:-Checks that there are ECB_SIZE bytes available in memory.

[0143] -Creates an ECBoECB_Pointer[0.. DM_POINTER_SIZE-1] = Requests ECB_SIZE bytes of memory.

[0144] -Searches for a non-existing Mutex_ID to assign it to the new mutex.

[0145] -Initializes ECBoECB.Event_ID = Mutex_IDoECB.Control_Stat = OFFoLast_ECB.Next_ECB[0.. DM_POINTER_SIZE-1] = ECB_Pointer[0.. DM_POINTER_SIZE-1]oLast_ECB[0.. DM_POINTER_SIZE-1] = ECB_Pointer[0.. DM_POINTER_SIZE-1]-Searches for TIB, using Task_IDoTIB_Pointer = FIRST_TIB_POINTERoChecks that TIB.Task_ID = Task_IDolf not, TIB_Pointer = TIB.Next_TCBoRepeat search until appropriate TIB is found.

[0146] -Updates TIBoTIB.Mutex_ID = Mutex_ID

[0147] *SIOS_Set_Mutex(Mutex_ID)* Updates the mutex"s state.

[0148] Parameters:-Mutex_ID, in System_Pipe[0]Returns: (nothing)Actions:-Searches for ECBoECB_Pointer = FIRST_ECB_POINTERoChecks that ECB.Event_ID = Event_IDolf not, ECB_Pointer = ECB.Next_ECBoRepeat search until appropriate ECB is found.

[0149] -Updates ECBoECB.Control_Status = ON

[0150] *SIOS_Clear_Mutex(Mutex_ID)* Updates the mutex's state.

[0151] Parameters:-Mutex_ID, in System_Pipe[0]Returns: (nothing)Actions:-Searches for ECBoECB_Pointer = FIRST_ECB_POINTERoChecks that ECB.Event_ID = Event_IDoIf not, ECB_Pointer = ECB.Next_ECBoRepeat search until appropriate ECB is found.

[0152] -Updates ECBoECB.Control_Status = OFF

[0153] *SIOS_Create_Pipe(Pipe_Size)* Creates a PCB in memory, reserves memory space for the Pipe, and updates the necessary pointers.

[0154] Parameters:-Pipe_Size, in System_Pipe[0]Returns:-Pipe_ID, in System_Pipe[1] Actions:-Checks that there are PCB_SIZE bytes available in memory.

[0155] -Checks that there are Pipe_Size bytes available in memory.

[0156] -Creates a PCBoPCB_Pointer[0.. DM_POINTER_SIZE-1] = Requests PCB_SIZE bytes of memory-Searches for a non-existent Pipe_ID-Initializes PCBoPCB.Pipe_Base_Address[0.. DM_POINTER_SIZE-1] = Requests Pipe_Size bytes of memory.

[0157] oPCB.Pipe_ID = Pipe_IDoPCB.Pipe_Size = Pipe_SizeoLast_PCB.Next_PCB[0.. DM_POINTER_SIZE-1] = PCB_Pointer[0.. DM_POINTER_SIZE-1]oLast_PCB[0.. DM_POINTER_SIZE-1] = PCB_Pointer[0.. DM_POINTER_SIZE-1]

[0158] *SIOS_Set_Pipe_Status(Pipe_ID, Pipe_Status)* Updates a Pipe"s state indicator.

[0159] Parameters:-Pipe_ID, in System_Pipe[0]-Pipe_Status, in System_Pipe[1]Returns: (nothing)Actions:-Searches for ECBoECB_Pointer = FIRST_ECB_POINTERoChecks that ECB.Event_ID = Pipe_IDoIf not, ECB_Pointer = ECB.Next_ECBoRepeat search until an appropriate ECB is found.

[0160] -Updates stateoECB.Event_Control = Pipe_Status

[0161] *SIOS_Get_Pipe_Status(Pipe_ID)* Updates a Pipe"s state indicator.

[0162] Parameters:-Pipe_ID, in System_Pipe[0]Returns:-Pipe_Status, in System_Pipe[1] Actions:-Searches for ECBoECB_Pointer = FIRST_ECB_POINTERoChecks that ECB.Event_ID = Pipe_IDoIf not, ECB_Pointer = ECB.Next_ECBoRepeat search until an

appropriate ECB is found.

[0163] -Read stateoPipe_Status = ECB.Event_Control

[0164] *SIOS_Send_Byte_To_Pipe(Pipe_ID, Pipe_Index, Byte_Value)* Fills in the contents of a Pipe with one byte.

[0165] Parameters:-Pipe_ID, in System_Pipe[0]-Pipe_Index, in System_Pipe[1]-Byte_Value, in System_Pipe[2]Returns: (nothing)Actions:-Searches for PCBoPCB_Pointer = FIRST_PCB_POINTERoChecks that PCB.Pipe_ID = Pipe_IDoIf not, PCB_Pointer = PCB.Next_PCBoRepeat search until an appropriate PCB is found.

[0166] -Checks that the Pipe is not in use.

[0167] oPCB.Pipe_Control <> IN_USE-Updates Pipeo(PCB.Pipe_Base_Address + Pipe_Index) = Byte_ValueoPCB.Pipe_Control = IN_USE

[0168] *SIOS_Read_Byte_From_Pipe(Pipe_ID, Pipe_Index)* Retrieves a byte from the contents of a Pipe.

[0169] Parameters:-Pipe_ID, in System_Pipe[0]-Pipe_Index, in System_Pipe[1]Returns: -Byte, in System_Pipe[2]Actions:-Searches for PCBoPCB_Pointer = FIRST_PCB_POINTERoChecks that PCB.Pipe_ID = Pipe_IDoIf not, PCB_Pointer = PCB.Next_PCBoRepeat search until an appropriate PCB is found..

[0170] -Read PipeoByte = (PCB.Pipe_Base_Address + Pipe_Index)

[0171] *SIOS_Post_Port_Request(Request_Port_Mask)* Sends message requesting exclusive access to digital Input/Output portsParameters: -Request_Port_Mask, in System_Pipe [0..PORT_BYTES-1]Returns: (nothing)Actions:-Searches for PIB of the Port_Manager taskoPCB_Pointer = PORT_MANAGER_PIPE_BASE_ADDRESS-Checks that the Pipe is not in use.

[0172] -Updates Pipe"s control state.

[0173] oPCB.Pipe_Control = PORT_REQUEST_POSTED

[0174] *SIOS_Post_Port_Release(Release_Port_Mask)* Sends message to release a digital I/O port.

[0175] Parameters:-Release_Port_Mask, in System_Pipe[0..PORT_BYTES-1]Returns: (nothing)Actions:-Searches for PIB of the Port_Manager taskoPCB_Pointer = PORT_MANAGER_PIPE_BASE_ADDRESS-Checks that the Pipe is not in use.

[0176] -Updates Pipe"s control state.

[0177] oPCB.Pipe_Control = PORT_RELEASE_POSTED

[0178] *SIOS_Post_Port_Input_Setting(Input_Port_Mask)* Sends request message to digital I/O ports selected as input ports so that port events may be generated in case of a change in their state.

[0179] Parameters:-Input_Port_Mask, in System_Pipe[0..PORT_BYTES-1]Returns: (nothing) Actions:-Searches for PIB of the Port_Manager task.

[0180] oPCB_Pointer = PORT_MANAGER_PIPE_BASE_ADDRESS-Checks that Pipe is not in use.

[0181] -Updates Pipe"s control state.

[0182] oPCB.Pipe_Control = PORT_INPUT_SETTING_POSTED

[0183] *SIOS_Post_Port_Idle_Setting(Idle_Port_Mask)* Sends request message to digital I/O ports selected as input ports so that port events may be generated in case of a change in their state. A change is detected by comparing the current state of the port against the IDLE value that exists on the Pipe.

[0184] Parameters: -Idle_Port_Mask, in System_Pipe[0..PORT_BYTES-1]Returns: (nothing) Actions:-Searches for PIB of the Port_Manager task.

[0185] oPCB_Pointer = PORT_MANAGER_PIPE_BASE_ADDRESS-Checks that the Pipe is not in use.

[0186] -Updates Pipe"s control stateoPCB.Pipe_Control = PORT_IDLE_SETTING_POSTED

[0187] *SIOS_Post_Port_Data (Port_Data_Value)* Sends message requesting that data be put into digital I/O ports.

[0188] Parameters: -Port_Data_Value, in System_Pipe[0..PORT_BYTES-1]Returns: (nothing)

Actions:-Searches for PIB of the Port_Manager task.

[0189] oPCB_Pointer = PORT_MANAGER_PIPE_BASE_ADDRESS-Checks that Pipe is not in use.

[0190] -Updates Pipe"s control state.

[0191] -PCB.Pipe_Control = PORT_DATA_POSTED

[0192] *SIOS_Post_Port_Read ()* Sends message requesting that data be read from digital I/O ports.

[0193] Parameters: (none)Returns:- Port_Data_Value, in System_Pipe[0..PORT_BYTES-1] Actions:-Searches for PIB of Port_Manager task.

[0194] oPCB_Pointer = PORT_MANAGER_PIPE_BASE_ADDRESS-Checks that the Pipe is not in use.

[0195] -Updates Pipe"s control state.

[0196] oPCB.Pipe_Control = PORT_READ_POSTED

[0197] *SIOS_Check_Free_Memory_Block(Block_Size)* Checks the availability of a continuous data memory block.

[0198] Parameters:-Block_Size, in System_Pipe[0]Returns:-YES / NO, in System_Pipe[1] Actions:-Compares Block_Size with Memory_Remainder

[0199] *SIOS_Read_Program_Memory(PM_Pointer, Block_Size, Pipe_ID)* Reads block of program memory and puts it into a Pipe.

[0200] Parameters:-PM_Pointer, in System_Pipe[0.. PM_POINTER_SIZE-1]-Block_Size, in System_Pipe[PM_POINTER_SIZE]-Pipe_ID, in System_Pipe[PM_POINTER_SIZE+1]Returns: (nothing)Actions:-Searches for PCBoPCB_Pointer = FIRST_PCB_POINTERoChecks that PCB.Pipe_ID = Pipe_IDoIf not, PCB_Pointer = PCB.Next_PCBoRepeat search until an appropriate PCB is found.

[0201] -Checks that there is enough free space in the Pipe to store the data block.

[0202] oPCB.Pipe_Size >= Block_Size-Determines the type of memory to be read

depending on the range of memory where the memory block to be read resides.

[0203] -Select the appropriate reading procedure, according to the type of memory to be read.

[0204] -Read bytes into program memory and put them into a Pipe.

[0205] *SIOS_ Logic_Function(Logic_Function, Buffer_Size, BufferA_Pointer, BufferB_Pointer)* Carries out the specified logic function between two data buffers of same length.

[0206] Parameters:-Logic_Function, in System_Pipe[0]-Buffer_Size, in System_Pipe[1]-BufferA_Pointer, in System_Pipe[2..1+DM_POINTER_SIZE]-BufferB_Pointer, in System_Pipe[2+DM_POINTER_SIZE.. 1+2*DM_POINTER_SIZE]Returns:-Result is put into Buffer_AActions:-Determines what function is to be carried out.

[0207] -Reads bytes from both buffers.

[0208] -Executes function using read bytes as parameters.

[0209] -Stores results in the byte in buffer_A.

[0210] *SIOS_ ADD_Function(Buffer_Size, BufferA_Pointer, BufferB_Pointer)* Executes an arithmetic ADD between two data buffers of same length.

[0211] Returns:-Results goes into Buffer_AParameters:-Buffer_Size, in System_Pipe[0]-BufferA_Pointer, in System_Pipe[1.. DM_POINTER_SIZE]-BufferB_Pointer, in System_Pipe [DM_POINTER_SIZE+1..2* DM_POINTER_SIZE]Actions:-Reads bytes from each buffer, starting with the smallest index.

[0212] -Executes ADD between read bytes (taking CARRY into consideration)-Stores results in Buffer_A byte.

[0213] *SIOS_ SUB_Function(Buffer_Size, BufferA_Pointer, BufferB_Pointer)* Executes an arithmetic SUB (subtraction) between two data buffers of same length.

[0214] Parameters:-Buffer_Size, in System_Pipe[0]-BufferA_Pointer, in System_Pipe[1.. DM_POINTER_SIZE]-BufferB_Pointer, in System_Pipe[DM_POINTER_SIZE+1..2* DM_POINTER_SIZE]Returns:-Results goes into Buffer_AActions:-Reads bytes from each buffer, starting with the smallest index.

[0215] -Executes SUB between read bytes (taking CARRY into consideration)-Stores results in Buffer_A byte.

[0216] CONCLUSION, RAMIFICATIONS AND SCOPE OF INVENTION

[0217] Thus, the reader will see that SIOS is a multitasking operating system capable of operating on low processing power platforms, such as 8-bit microprocessors. In addition, the memory resources it requires for full functionality are very reduced so as to permit the concurrent execution of operating system functions and application tasks within the same microprocessor memory space. The simplicity of presented algorithms facilitates the ready implementation of SIOS on several different microprocessor platforms.

[0218] While our above description contains many details, these should not be construed as limitations to the scope of the invention, but rather as an exemplification of one preferred embodiment thereof. Obviously, modifications and alterations will occur to others upon a reading and understanding of this specification. For example, a higher degree of robustness can be achieved using a dynamic task priority scheme so that low priority tasks need not wait for a long time even if high-priority tasks are present. The dynamic priority scheme can augment the priority of low priority tasks as wait time increases. Eventually, low priority tasks will acquire higher priorities and receive control.

[0219] In addition, the OS can be further equipped with a high priority task that implements an external communication buffer that allows the existence of a command line style-like interface. Using such interface, a user could tell the OS to carry out specific actions, such as task activation and deactivation, read memory blocks, update task program code, monitoring of system error flags, modify the state of an event, and many others. The description above is intended, however, to include all such modifications and alterations insofar as they come within the scope of the appended claims or the equivalents thereof.